Code Generation for Parallel Execution of a Class of Irregular Loops on Distributed Memory Systems

Mahesh Ravishankar*, John Eisenlohr*, Louis-Noël Pouchet*, J. Ramanujam†, Atanas Rountev*, P. Sadayappan*
*The Ohio State University, †Louisiana State University
Email: *{ravishan,eisenloh,pouchet,rountev,saday}@cse.ohio-state.edu,†jxr@ece.lsu.edu

***Abstract*—Parallelization and locality optimization of affine loop nests has been successfully addressed for shared-memory machines. However, many large-scale simulation applications must be executed in a distributed-memory environment, and use irregular/sparse computations where the control-flow and array-access patterns are data-dependent.**

In this paper, we propose an approach for effective parallel execution of a class of irregular loop computations in a distributed-memory environment, using a combination of static and run-time analysis. We discuss algorithms that analyze sequential code to generate an *inspector* and an *executor*. The inspector captures the data-dependent behavior of the computation in parallel and without requiring complete replication of any of the data structures used in the original computation. The executor performs the computation in parallel. The effectiveness of the framework is demonstrated on several benchmarks and a climate modeling application.

I. Introduction

Automatic parallelization and locality optimization of affine loop nests have been addressed for shared-memory multi-processors and GPUs with good success [4], [7], [8], [16], [29], [30]. However, many large-scale simulation applications must be executed in a distributed-memory environment, using irregular or sparse computations where the control-flow and array-access patterns are data-dependent. A common form of sparsity and unstructured data in scientific codes is via indirect array accesses, where elements of one array are used as indices to access elements of another array. Further, multiple levels of indirection may be used for array accesses. Virtually all prior work on polyhedral compiler transformations is inapplicable in such cases.

We propose a framework for automatic parallelization and distributed-memory code generation for an extended class of affine computations that allow some forms of indirect array accesses. The class we address is prevalent in many scientific/engineering domains and the paradigm for its parallelization is often called the inspector/executor (I/E) [38] approach. The I/E approach uses (1) a so-called *inspector* code that examines some data that is unavailable at compile time but is available at the very beginning of execution (e.g., the specific inter-connectivity of the unstructured grid representing an airplane wing's discretized representation) to construct distributed data structures and computation partitions, and (2) an *executor* code that uses data structures generated by the inspector to execute the desired computation using parallelism.

The I/E approach has been well known in the high-performance computing community, since the pioneering work of Saltz and coworkers [38] in the late eighties. The approach is routinely used by application developers for manual implementation of message-passing codes for unstructured grid applications. However, only a very small number of compiler efforts (that we detail in Sec. VII) have been directed at generation of parallel code using the approach. In this paper, using the I/E paradigm, we develop an automatic parallelization and code generation infrastructure for an extended class of affine loops, targeting a distributed-memory message passing parallel programming model. This paper makes the following contributions:

- It presents a transformation system for effective automatic parallelization and distributed-memory code generation for an extended class of affine programs;
- It develops an efficient approach for generating parallel code with a lower degree of indirect array access than any previously proposed algorithms for the class of computations handled; and
- It presents experimental results on the use of the approach to develop a complex parallel application, with performance approaching that of manual parallelization implemented by expert application developers.

The rest of the paper is organized as follows. Section II describes the class of extended affine computations that we address, along with a high-level overview of the approach to code transformation. Section III provides details of the approach to generate computation partitions using a hypergraph that models the affinity of loop iterations to data elements accessed. The algorithms for generation of inspector and executor code are provided in Section IV. Section V elaborates on how the need for inspector code can be optimized away for portions of the input code that are strictly affine. Experimental results using four kernels and one significant application are presented in Section VI. Related work is discussed in Section VII and conclusions stated in Section VIII.

II. Overview

This section outlines the methodology for automatic parallelization of the addressed class of applications. Listing 1 shows two loops from a conjugate-gradient iterative sparse linear systems solver, an example of the class of computations targeted by our approach. Loop `k` computes the values of `x`. In loop `i`, vector `y` is computed by multiplying matrix `A` and vector `x`. Here `A` uses the Compressed Sparse Row (CSR) format, a standard representation for sparse matrices. For a

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00©2012 IEEE

```
1  while( !converged ){
2    /* Other computation; not shown */
3    for( k = 0 ; k < n ; k++ )
4      x[k] = ...
5    /* Other computation; not shown */
6    for( i = 0 ; i < n ; i++ )
7      for( j = ia[i] ; j < ia[i+1] ; j++ ){
8        xindex = col[j];
9        y[i] += A[j]*x[xindex];
10     }
11   /*Other computation; not shown */
12 }
```

Listing 1. Sequential conjugate gradient computation.

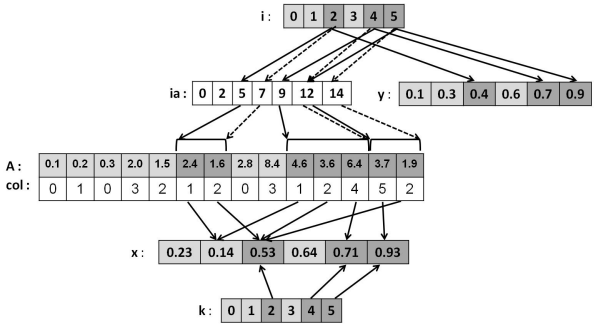


Fig. 1. Control flow and data-access patterns of iteration 2, 4, and 5, of loops i and k mapped to process 0.

sparse matrix with `n` rows, array `ia` is of size `n+1` and its entries point to the beginning of a consecutive set of locations in `A` that store the non-zero elements in row `i`. For `i` in [`0`,`n-1`], these non-zero elements are in `A[ia[i]]`, ..., `A[ia[i+1]-1]`. Array `col` has the same size as `A`, and for every element in `A`, `col` stores its column number in the matrix.

Figure 1 shows sample values for all arrays in the computation. The bounds of loop `j` depend on values in `ia`, and the elements of `x` accessed for any `i` depend on values in `col`. Such arrays that affect the control-flow and array-access patterns are referred to as *indirection arrays*. All other arrays will be referred to as *data arrays* (`x`, `y`, and `A`).

The goal is to parallelize the computation by partitioning the iterations of loops `i` and `k` among a set of given processes. Suppose that iterations 2, 4, and 5 (shown in dark gray) are chosen to be executed on process 0, and the remaining ones (shown in light gray) on process 1. As discussed below, the choice of this partitioning is done at run time with the help of a hypergraph partitioner. Figure 2 illustrates the details of this partitioned execution; these details will be elaborated shortly.

We present a source-to-source transformation scheme that (1) generates code to analyze the computation at run time for partitioning the iterations, as well as data, among processes, (2) generates local data structures needed to execute the partitioned iterations in a manner consistent with the original computation, and (3) executes the partitions on multiple processes. The code that performs the first two steps is commonly referred to as an *inspector*, with the final step performed by an *executor*. Listing 2 shows the latter for the running example.

Targeted computations. We target a class of computations that are more general than *affine* computations. In affine codes, the loop bounds, conditionals of `if`s, and array-access expressions

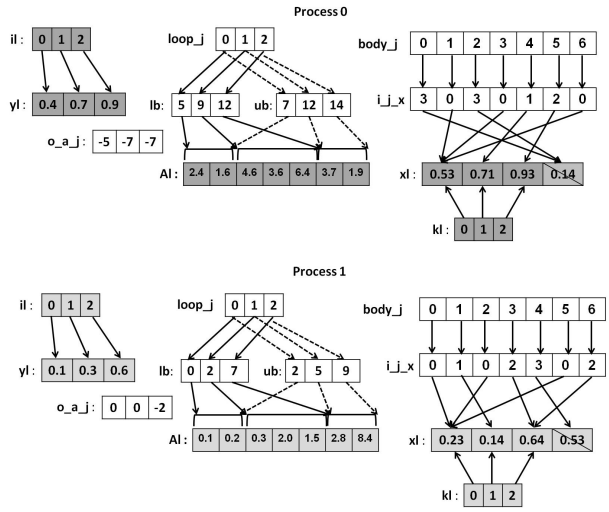


Fig. 2. Transformed iteration and data view.

are affine functions of loop iterators and program parameters. For such codes, the control-flow and data-access patterns can be fully characterized at compile time.

Consider loop `i` in Listing 1. The bounds of loop `j` depend on `ia`, and accesses to `x` depend on `col`. During analysis of loop `i`, affine techniques have to be conservative and over-approximate the data dependences and control flow. We target a generalized class of computations, in which loop bounds, `if` conditionals, and array-access expressions are affine functions of iterators, parameters, and values stored in indirection arrays. Further, values in these indirection arrays may themselves be accessed through other indirection arrays.

Within this extended class, we target loops that are parallel, except for loop-carried dependences due to reductions of scalars or array elements using operators which are associative and commutative. For any scalar or array element participating in such reductions, its value should not be used for any other computation within the loop body. Values of indirection arrays must not be modified within the loop. Loops that satisfy these properties will be referred to as *partitionable loops*. Loops `i` and `k` in Listing 1 are partitionable. For code transformations, only partitionable loops not nested within each other are considered. A formal definition of partitionable loops can be found in [34].

The proposed framework is well suited for computations that have a sequence of partitionable loops enclosed within an outer sequential loop (usually a time-step loop or a convergence loop), such that the control-flow and array-access patterns are not modified within the sequential loop. Such computations are common in many scientific and engineering domains. Furthermore, with this code structure, the inspector can be hoisted out of the outer loop.

Partitioning the iterations. In Listing 1, there exists a producer-consumer relationship between the two loops due to array `x`. In a parallel execution of both loops, communication would be required to satisfy this dependence. The volume of communication depends on the partitioning of the iterations. The process of computing these partitions (Section III) may

```
1  /* Inspector code to generate o_a_j, i_x_j, lb,*/
2  /* ub, xl, yl, Al and to compute nl; not shown */
3  while( !converged ){
4    /* Other computation; not shown */
5    /* Update ghosts */
6    body_k = 0;
7    for( kl = 0 ; kl < nl ; kl++ ){
8      xl[kl] = ...; body_k++; }
9    /* Update owners */
10   /* Other computation; not shown */
11   /* Update ghosts */
12   body_i = 0; loop_j = 0 ; body_j = 0;
13   for( il = 0 ; il < nl ; il++ ) {
14     offset_a_j = o_a_j[loop_j];
15     for( j = lb[loop_j] ; j < ub[loop_j] ; j++ ){
16       yl[il] += Al[j+offset_a_j]*xl[i_x_j[body_j]];
17       body_j++;   }
18     loop_j++; body_i++;  }
19   /* Update owners */
20 }
```

Listing 2. Parallel conjugate gradient computation.

result in the iterations mapped to each process not being contiguous. They will be renumbered to be a contiguous sequence starting from 0. For example, iterations 2, 4, and 5 of loop `i`, when assigned to process 0, are renumbered 0–2 (shown as the values of local iterator `il` in Figure 2).

Bounds of inner loops. The control-flow in the parallel execution needs to be consistent with the original computation. As discussed earlier, the loop bounds of inner loops depend on values stored in read-only indirection arrays, loop iterators, or fixed-value parameters. Therefore, these bounds can be precomputed by an inspector and stored in arrays in the local data space of each process. The sizes of these arrays would be the number of times an inner loop is invoked on that process. For example, in Figure 1, for iterations mapped to process 0, inner loop `j` is invoked once in every iteration of loop `i`. Two arrays of size 3 would be needed to store the bounds of the loop on process 0 (shown as `lb` and `ub` in Figure 2). Conditionals of `if`s are handled similarly, by storing their values in local arrays.

Partitioning the data. Once the iterations have been partitioned, the data is partitioned such that each process has local arrays to store all the data needed to execute its iterations without any communication within the loop. In Figure 2, `yl`, `Al`, and `xl` are the local arrays on process 0 for `y`, `A`, and `x`.

The same array element may be accessed by multiple iterations of the partitionable loop, which might be executed on different processes. Consider Figure 1, where `x[1]` and `x[2]` are accessed by both processes and are replicated on both, as shown in Figure 2. One of the processes is chosen as the *owner* of the data, and the location of the data on other processes is treated as a *ghost location*. For example, `x[2]` is owned by process 0, but process 1 has a ghost location for it. The ghost locations and owned locations together constitute the local copy of a data array on a process.

Ghost elements for arrays that are only read within the partitionable loop are set to the value at the owner before the start of the loop. Ghost locations for arrays whose values are updated within the loop are initialized to the identity element of the update operator (0 for "+=", 1 for "*="). After the loop, these elements are communicated to the owner where the values from all ghost locations are combined. Therefore, the computation model is not strictly *owner-computes*. Since all update operations are associative and commutative, all iterations of the loop in the transformed version can be executed without any communication.

Data accesses in the transformed code. The data-access patterns of the original computation need to be replicated in the transformed version. Consider expression `col[j]` used to access `x` in Listing 1. Since `xl` is the local copy of `x` on each process, all elements of `x` accessed by a process are represented in `xl`. To access the correct elements in `xl`, array `col` could be replicated on each process, and a map could be used to find the location that represents the element `col[j]` of `x`. Such an approach would need a map lookup for every memory access and would be prohibitively expensive.

Similar to loop bounds, array-access expressions depend only on values stored in read-only indirection arrays, loop iterators, and constant parameters. Values of these expressions can be inspected and stored in arrays allocated in the local memory of each process. Further, the values stored are modified to point to corresponding locations in the local data arrays. The size of the array would be the number of times the expression is evaluated on a particular process. For example, the value of `col[j]` in Listing 1 is evaluated for every iteration of loop `j`. From Figure 1, for iterations of `i` mapped to process 0, the total number of iterations of loop `j` executed is $2+3+2=7$. Therefore an array `i_x_j` of size 7 on process 0 is used to "simulate" the accesses to `x` due to expression `col[j]`.

Optimizing accesses from inner loops. The procedure described earlier would result in another array (of the same size as `i_x_j`) to represent the access `A[j]`. To reduce the memory footprint, we recognize that access expression `j` results in contiguous accesses to elements of `A`, for every execution of loop `j`. If the local array is such that elements that were accessed contiguously in the original data space remain contiguous in the local data space of each process, it would be enough to store (in an additional array) the translated value of the access expression for only the first iteration of the loop. The rest of the accesses could be derived by adding to this value the value of the iterator, subtracted by the lower bound. The size of the array to hold these values is the number of times the loop is invoked. For example, for `A[j]`, an array `o_a_j` of size 3 is used on process 0 to store the accesses from the first iterations of the 3 invocations of loop `j`.

This optimization is applicable to all array-access expressions that are unit-stride with respect to a surrounding loop that is not a partitionable loop. Accesses from iterations of a partitionable loop mapped to a process are not necessarily contiguous with respect to the original computation.

Optimizing accesses from the partitionable loop. For cases where elements of an array were accessed at unit-stride with respect to the partitionable loop in the original computation, it is desirable to maintain unit-stride in the transformed code as well. This can be achieved by placing contiguously in local memory all elements of the array accessed by the successive iterations on a process. For example, if iterations 2, 4, and 5

of loop k are mapped to process 0, elements of array xl can be accessed by using iterator kl if xl[0-2] correspond to x[2], x[4], and x[5]. The same could be done for y[i] in Listing 1 since there are no other accesses to it.

If the same array is accessed by an expression that is unit-stride with respect to an inner loop (as described earlier), the ordering of elements required to maintain the unit-stride in the transformed code may conflict with the ordering necessary to maintain unit-stride with respect to a partitionable loop. In such cases, the accesses from the partitionable loop are not optimized. If multiple partitionable loops access an array with unit-stride, to optimize all the accesses the loops must be partitioned in a similar way in order to obtain a consistent ordering of the array elements (Section IV-A).

Executor code. To execute the original computation on each process, code is transformed such that the modified code accesses local arrays for all data arrays, and uses values stored in local arrays for loop bounds, conditionals, and array-access expressions. Listing 2 shows the modified code obtained from Listing 1. Loop bounds of partitionable loops are based on the number of iterations nl that are mapped to a process. The loop bounds of loop j are read from arrays lb and ub. Accesses to local arrays xl and Al are determined by values in arrays i_x_j and o_a_j. Also, communication calls are inserted to satisfy the producer-consumer relationship due to array x.

III. Partitioning the Computation

The computation is partitioned by considering the iteration-data affinity. To model this affinity, we use a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ where $\mathcal{V}$ is the set of vertices and $\mathcal{N}$ is the set of nets [43]. Iterations of all partitionable loops are represented by separate vertices $i \in \mathcal{V}$. Each accessed element of a data array is represented by a net $j \in \mathcal{N}$, whose pins are the iterations that access the data element. The hypergraph is subjected to a multi-constraint partitioning to (1) partition the iteration of each of the partitionable loop in a load-balanced manner, and (2) minimize communication required for producer-consumer relationships between partitionable loops.

Achieving load balancing. Each vertex is associated with a vector of weights $\vec{w_i}$ of size equal to the number of partitionable loops. A vertex that represents an iteration of partitionable loop k has the k-th element as 1, with all other elements being 0. The weight of a set of vertices P is defined as $W_P = \sum_{i \in P} \vec{w_i}$. If P_n (where $n \in [0, N)$) are the partitions generated, load balance for every partitionable loop is achieved by applying the constraint $P_n \leq P_{avg}(1+\epsilon)$, where $P_{avg} = P_{\mathcal{V}}/N$; ϵ is the maximum load imbalance tolerated.

Minimizing communication. Each net in $\mathcal{N}$ is also associated with a weight c_j, whose value is the same as the size of the data represented by the net. For each partition P_n, the set of nets that have pins in it can be divided into two disjoint subsets. Nets that have pins only in P_n are *internal* nets, I_n and nets with pins in other partitions are *external* nets, E_n. Each external net represents a data element that is accessed by more than one process. One of the partitions is the owner of the data, and the other partitions have corresponding ghosts. To minimize communication, the number of ghost cells needs to be minimized, along with the number of partitions λ_j that have a ghost for the data element represented by $j \in \mathcal{N}$. This is achieved by minimizing the *cut-size* Π_n for each partition defined as $\Pi_n = \sum_{j \in E_n} c_j(\lambda_j - 1)$. The hypergraph is subjected to a min-cut partitioning, under the load-balance constraints specified above [9].

IV. Inspector: Functionality and Code Generation

This section describes the run-time inspector analysis required to create the iteration partitions (along with the required data) for each process. The process has three phases.

- Phase I: Build and partition the hypergraph by analyzing the data elements touched by the iterations of all partitionable loops; allocate local copies for all data arrays based on the iterations assigned to each process.
- Phase II: Compute the sizes of the arrays needed to replicate the control-flow and array-access patterns.
- Phase III: Populate these arrays with appropriate values.

A. Phase I: Hypergraph Generation

Run-time functionality. The inspector analyzes the computation and generates the corresponding hypergraph. For Listing 1, a portion of the inspector that generates the hypergraph is shown in Listing 3. The inspector code contains only statements from the original computation that affect the control flow and array accesses.

The inspector (for the sake of analysis) starts with the assumption that all arrays are block-partitioned across the processes. Each process analyzes a block-partitioned subset of the original iteration (represented by [kstart,kend) for loop k and [istart,iend) for loop i) and therefore computes only a part of the hypergraph. For each iteration of the partitionable loop executed on a process, a vertex is added to represent it in the hypergraph, by calling AddVertex. For every data array element that is accessed by this iteration, the vertex is added as a pin to the corresponding net. For example, AddPin(id_y,i,vi,1) adds vertex vi as a pin to the net for the i-th element of array y (id_y is a unique id for y). The last argument specifies that the element is accessed by an expression with a unit stride.

Since arrays are block-partitioned, it might not be possible to evaluate each array-access expression since values in indirection arrays might not be local to the process. Thus, every access to an indirection array is guarded by the function is_known which returns true if the value needed is known on the current process and false otherwise, with the element flagged as being requested. After the block of iterations have been analyzed, all outstanding requests are serviced. On re-analyzing these iterations, is_known for those elements would evaluate to true, and the value can be obtained via function get_elem. Repeated analysis is performed until is_known returns true for all accessed elements. In this phase, there is no communication due to the values of the data array elements, since these values are not used to index other arrays. Multiple levels of indirection are handled through successive execution

```
1  do{ for( k = kstart ; k < kend ; k++ ){
2        AddVertex(id_k,vk);
3        AddPin(id_x,k,vk,1); ...}
4      for( i = istart ; i < iend ; i++ ){
5        AddVertex(id_i,vi);
6        if(is_known(id_ia,i) && is_known(id_ia,i+1))
7          for( j = get_elem(id_ia,i) ;
8              j < get_elem(id_ia,i+1) ; j++ ){
9            AddPin(id_y,i,vi,1); AddPin(id_A,j,vi,1);
10           if( is_known(id_col,j) )
11               xindex = get_elem(id_col,j);
12           if( is_known(id_xindex) )
13             AddPin(id_x,xindex,vi,0);     }
14     }while( DoneGraphGen() );
```

Listing 3. Phase I of the inspector.

of the outer blocked-partitioned loop, as shown in Listing 3.

The portions of the hypergraph built by each process are combined to compute the complete iteration-to-data affinity. The hypergraph is partitioned P ways as described in Section III where P is the number of processes. Each process is assigned a unique partition representing the iterations to be executed on it. The iterations are renumbered such that they form a contiguous set on each process, while maintaining the relative ordering of the iterations mapped to that process.

Code generation at compile time. The inspector code that achieves the functionality described above (e.g., the code in Listing 3) is generated automatically by the compiler. To analyze the control-flow and array-access patterns of the original computations, all loops and conditional statements are included in the inspector code with certain modifications, as discussed below. Since scalars might be involved in loop bounds, conditionals, and index expressions of arrays, all assignments to such scalars are also included. For example, `xindex` in Listing 1 is used to access array `x`. Therefore, statement `xindex=col[j]` has to be executed by the inspector to capture the elements of `x` accessed. Scalars that are used (directly or transitively) to determine control flow or array-access expressions inside a partitionable loop (referred to as *inspected scalars*) are handled in this manner.

To ensure that values not yet known on a process are not used, for all array elements and inspected scalars, boolean state variables are maintained. Loops/branches are executed only if all data accessed in the bounds/conditionals are known on a process, as illustrated by the calls to `is_known`. Similarly, inspected scalars are assigned values only if all elements on the right-hand side are known.

To support the optimizations of accesses from partitionable loops as discussed in Section II, it might be necessary to ensure that multiple partitionable loops are partitioned the same way. To enforce this, the loop bounds of all such partitionable loops are checked at compile time. If they are the same, `AddVertex` would map corresponding iterations of all these loops to the same vertex. In cases where the loop bounds are not the same, the accesses to data arrays are not optimized.

B. Initializing Local Arrays

After partitioning the hypergraph, Phase I of the inspector partitions the data. For a net that has all its pins in the same partition, the corresponding data element is assigned to the same process. If a net has pins in different partitions, the element is assigned to the process that executes the majority of the iterations that access this data. All other processes have a ghost location for that element. The local copy of the array consists of the elements that a process owns and the ghosts locations for elements owned by other processes. This is done for all data arrays in the computation. For example, arrays `yl`, `xl` and `Al` of Listing 2 are allocated at this time.

A compile-time analysis determines expressions that result in unit-stride accesses to a data array due to a surrounding loop, with the conflict between accesses from a partitioned loop and from an inner loop resolved statically. Elements accessed by such expressions (known at run time using the last argument of `AddPin`) are laid out first in increasing order of their original position, followed by all other elements of the array accessed. This scheme maintains the contiguity of accesses within inner loops in the transformed code and within partitionable loops when they do not conflict with the former.

C. Phase II: Computing the Sizes of Local Access Arrays

Run-time functionality. The next step is to determine the sizes of arrays used to (1) store loop bounds of inner loops, (2) store the result of conditionals, and (3) store the indices of accessed data array elements. The size of these arrays depends on the expressions they represent. Every array-access expression is analyzed at compile time to determine if the access is unit-stride with respect to a surrounding loop. For such an expression, the size of the array needed would be the same as the number of invocations of the corresponding loop. For expressions that are not unit-stride with respect to any surrounding loop, loop-invariant analysis is performed to determine the innermost loop with respect to which the value of the expression changes. In the worst case, this might be the immediately surrounding loop. The size of the array needed to represent these expressions is the total number of iterations of this loop across all iterations of the partitioned loop.

The sizes of arrays that store the bounds of an inner loop are the same as the number of invocations of the loop. For an array needed to store the values of a conditional, the number of times the `if` statement is executed should be counted. Listing 4 shows the code for this phase of the inspector for the running example. The number of invocations of inner loop `j` is tracked via counter `loop_j`. Counters `body_*` track the total number of times a loop body is executed. In addition, this phase ensures that a process has all values of indirection arrays needed to analyze the iterations mapped to it.

Each process executes only the iterations mapped to it after partitioning, which may be different from those analyzed by this process when building the hypergraph. As in Section IV-A, all inner loops, conditionals, and statements are guarded to check if the values of inspected scalars and indirection array elements have been determined. Again, this phase is completed only after all levels of indirections have been resolved.

Code generation at compile time. The code generation for this phase is similar to Phase I. Only statements that affect the control-flow and array-access patterns are considered. The

```
1  do{
2    body_i = 0 ; loop_j = 0 ; body_j = 0; body_k = 0;
3    for( k = 0 ; k < n ; k++ )
4      if( home(id_k,k) == myid )
5        body_k++;
6    for( i = 0 ; i < n ; i++ )
7      if( home(id_i,i) == myid ){
8        if( is_known(id_ia,i) && is_known(id_ia,i+1)){
9          for( j = get_elem(id_ia,i) ;
10              j < get_elem(id_ia,i+1) ; j++ ){
11           if( is_known(id_col,j) )
12             xindex = get_elem(id_col,j);
13           body_j++;                              }
14         loop_j++;                              }
15       body_i++;                 }
16   }while( DoneCounters() );
```

Listing 4. Phase II of the inspector.

```
1  body_i = 0 ; loop_j = 0 ; body_j = 0; body_k = 0;
2  for( k = 0 ; k < n ; k++ )
3    if( home(id_k,k) == myid )
4      body_k++;
5  for( i = 0 ; i < n ; i++ )
6    if( home(id_i,i) == myid ){
7      lb_j[loop_j] = get_elem(id_ia,i);
8      ub_j[loop_j] = get_elem(id_ia,i+1);
9      for( j = lb_j[loop_j] ; j < ub_j[loop_j] ; j++){
10       xindex = get_elem(id_col,j);
11       if( j == lb_j[loop_j] )
12         o_a_j[loop_j] = j;
13       i_x_j[body_j] = xindex; body_j++;          }
14     loop_j++;
15     body_i++;                }
```

Listing 5. Phase III of the inspector.

differences from Phase I are (1) the bounds of the partitionable loop are same as the original computation and its body is enclosed in an `if` statement that checks if the iteration is to be executed on the current process, and (2) statements to increment counters `loop_*` and `body_*` are introduced.

D. Phase III: Initializing Local Access Arrays

Run-time functionality. After allocation, the access arrays are initially populated with the sequence of values the corresponding expression evaluates to in the original computation. Listing 5 shows the code to do so, for the example in Listing 1.

For expressions that are unit-stride with respect to a surrounding inner loop, the element accessed by the first iteration of every invocation of the loop is stored in the array that represents the index expression. For all other expressions, the values for all iterations are stored in arrays. The value of the loop bounds of all inner loops and results of conditionals are also stored in arrays in this phase of the inspector.

Code generation at compile time. To generate the code for this phase, once again the partitionable loop is replicated as is, with the loop body enclosed within an `if` statement to analyze only the iterations mapped to the current process. The body of this new loop is generated by traversing the statements of the original partitionable loops, as shown in Algorithm 1.

Assignment Statements: On encountering such statements in the original AST, a corresponding statement is added to the AST of this phase of the inspector by Algorithm 1. Statements that are assignments to inspected scalars are replicated with references to indirection arrays replaced with calls to `get_elem` by *CovertArraysToFunctions*. For example, the right-hand side of `xindex=col[j]` in Listing 1 is modified

Algorithm 1: CodeToInitializeAccessArrays(s,A,$\mathcal{C}$,S,$\mathcal{A}_P$)

Input : s : Statement in the original AST; A : Access Arrays
$\mathcal{C}$: Counter Variables; S : Inspected Scalars
InOut: $\mathcal{A}_P$: AST of code to populate the access arrays

1 **begin**
2 **if** *s.LHS* $\notin S$ **then**
3 **foreach** $d \in$ *GetDataArrayRefExp(s)* **do**
4 a = AccessArray(A,d.Array,d.IndexExpr) ;
5 c = CounterVariable($\mathcal{C}$,d.IndexExpr) ;
6 l_P = CreateStoreAccessArray(a,c,d.IndexExpr) ;
7 **if** *IsUnitStride(d.IndexExpr)* **then**
8 l = GetLoop(c) ; l_P = CreateIfFirstIter(c,l_P) ;
9 $\mathcal{A}_P$.Append(l_P) ;
10 **else**
11 b = ConvertArraysToFunctions(s.RHS) ;
12 l_p = AssignmentStatement(s.LHS,b) ;
13 $\mathcal{A}_P$.Append(l_P) ;

to `get_elem(id_col,j)`. There is no need for any guards in this phase since the previous phase ensured that all values needed are known on the current process.

For all other assignment statements, function *GetDataArrayRefExp* returns the set of all expressions used to access data array elements. For every such expression, an assignment is generated by function *CreateStoreAccessArray* to store the value of the corresponding index expression, with all references to indirection arrays replaced with calls to `get_elem`.

Further, if the index expression is unit-stride with respect to a surrounding loop, the statement is enclosed within an `if` statement, generated by *CreateIfFirstIter*, which checks if the value of the iterator of the loop is same as that stored in the lower-bound array. For example, in Listing 5, `o_a_j` stores the value of expression `j` used to access array `A`, and is enclosed within an `if` statement that is true for the first loop iteration.

Having populated all access arrays with the original values of the expressions they represent, these values are now modified to point to the corresponding locations in the local copies of the arrays being accessed. For access arrays that represent expressions that are unit-stride with respect to a loop, the modified values are element-wise subtracted with the values stored in the lower-bound array of the loop. Adding the loop iterator value to this would point to the correct location.

Loops and conditionals: Corresponding to inner loops within the original partitionable loop, statements to store the value of the current upper/lower bounds of the loop are inserted in the AST. The loop itself is inserted after these statements, with the bounds modified to use these stored values. Conditional statements are treated similarly.

E. Executor Code

After all phases of the inspector, the loop iterations and data arrays have been partitioned among the processes. All access arrays have been initialized with values that point to the appropriate locations in the local arrays.

The executor code is similar to the original code. All counter variables are first reset to 0. The lower and upper bounds of the partitioned loops are set to 0 and the number of assigned

iterations, respectively.

Assignment Statements: For such a statement in the original code, the corresponding statement in the executor is generated as follows. Statements that write to inspected scalars are not inserted in the executor, since the control flow and array accesses are handled explicitly through arrays. For all other assignment statements, accesses to original data arrays are replaced with accesses to corresponding local arrays. The generated index expressions depend on the original index expressions. For those that are unit-stride with respect to a loop, the index expression is the sum of the iterator value and the value stored in the corresponding access array. For example, in Listen 2, the index to array `Al` is obtained by adding the offset stored in `o_a_j` to `j`. This index expression ensures unit-stride access to the array. This is important to enable subsequent SIMD optimizations and good spatial locality and cache prefetching. To the best of our knowledge, no previously proposed compiler approaches for I/E code generation ensure this highly-desirable property. The read from the access array is hoisted out of the corresponding loop (loop `j` for the example) since all iterations use the same offset.

For all other accesses, the access arrays store the index of the data array element that is to be accessed. In the executor, the original expression is replaced with a read from the corresponding access array.

Loops and conditionals: Loops and conditionals from the original computation are inserted into the AST of the executor, with loop bounds and conditional expressions modified to read from the arrays populated by Phase III of the inspector. Similar to Phases II and III of the inspector, counters are inserted into the AST of the executor to step through the values stored in the access arrays.

Once the executor code has been generated for all partitionable loops, communication calls to update the ghosts used within a loop are inserted before that loop in the executor AST. Communication calls to update the owner with values in all ghosts location are also inserted after the loop. The communication scheme used is described below.

F. Communication Between Processes

Elements of local data arrays consist of both owned and ghosts locations. Phase I of the inspector initializes them to their original values. To maintain correctness of the parallel execution, ghost cells for arrays that are read within a partitionable loop are updated before the start of the loop, and ghosts cells of arrays that are updated are communicated to the owner after the loop execution. For cases where partitionable loops assign values to array elements instead of updating them, the value of the ghost location from the process which executes the last iteration of the original computation is used to overwrite the value at the owner. The ID of the process can be computed by the inspector code while partitioning the computation.

The communication pattern used to update ghosts is similar to the MPI_Alltoallv collective. As the number of partitions increases, every process has to communicate with only a small number of other processes. Therefore, the communication costs are reduced by using one-sided point-to-point communication APIs provided by ARMCI [28].

Algorithm 2: CodeGenInspectorExecutor($\mathcal{P}$)

Input : $\mathcal{P}$: partitionable loop AST
Output: $\mathcal{A}_I$: Code for the inspector; $\mathcal{A}_E$: Code for the executor

begin
  S = GetInspectedScalars($\mathcal{P}$) ; $A_I = \phi$; $A_E = \phi$;
  $\mathcal{H}$ = CodeToCreateHypergraph($\mathcal{P}$,S); $\mathcal{A}_I$.Append($\mathcal{H}$) ;
  $\mathcal{A}_I$.Append(CodeToPartitionIterations($\mathcal{H}$)) ;
  $\mathcal{D}$ = CodeToAllocateLocalData($\mathcal{H}$) ;
  $\mathcal{C}$ = DeclareCounterVariables($\mathcal{P}$); $\mathcal{A}_I$.Append($\mathcal{C}$) ;
  $\mathcal{A}_C$ = CodeToGetAccessArraySize($\mathcal{P}$,S,$\mathcal{C}$) ;
  $\mathcal{I}$ = CodeToAllocateAccessArrays($\mathcal{P}$); $\mathcal{A}_C$.Append($\mathcal{I}$);
  $\mathcal{A}_P$ = CodeToInitializeArrays($\mathcal{P}$,S,$\mathcal{C}$,$\mathcal{I}$) ;
  $\mathcal{A}_P$.Append(CodeToRenumberAccessArrays($\mathcal{C}$,$\mathcal{D}$,$\mathcal{I}$)) ;
  $\mathcal{A}_I$.Append($\mathcal{A}_C$) ; $\mathcal{A}_I$.Append($\mathcal{A}_P$) ;
  $\mathcal{A}_E$ = GenerateExecutorCode($\mathcal{P}$,S,$\mathcal{C}$,$\mathcal{I}$,$\mathcal{D}$) ;

G. Putting Everything Together

The overall code generation scheme is shown in Algorithm 2. ASTs of the original partitionable loops are analyzed to find the inspected scalars. Next, the code to create the hypergraph is generated as described in Section IV-A, followed by the code to partition it. *CodeToAllocateLocalData* generates the code to partition the data arrays, as described in Section IV-B, and allocates local copies for these arrays.

CodeToGetAccessArraySize uses the steps described in Section IV-C to determine the sizes of the access arrays. Code to allocate the access arrays and arrays to store loop-bounds/conditional is then appended to the inspector code. *CodeToInitializeArrays* generates code to initialize these arrays, a part of which is presented in Algorithm 1. The values in the access arrays are then renumbered to point to the correct locations in the local arrays. The executor code is generated as outlined in Section IV-E.

V. Optimizations for Affine Code

When some partitionable loops in a program are completely affine, i.e., loop bounds and array-access expressions are strictly affine functions of surrounding loop iterators (and program parameters), the code generation described earlier is correct but introduces unnecessary overhead. For such loops, inspector code is unnecessary since control-flow and array-access patterns can be characterized statically. For example, if loop `i` in Listing 1 were of the form shown in Listing 6, where matrix `A` is dense and hence not stored in the CSR format, the computation can be analyzed statically.

A regular distribution of the iterations of partitionable loops (such as block, cyclic, or block-cyclic) is used instead of the hypergraph partitioning scheme. For data arrays accessed only through affine expressions, local copies can be computed as footprints of the partitioned iterations. Polyhedral code generation can be used since we only partition a single loop, with the number of iterations on a given process treated as a parameter. Ghosts can be computed as the intersection of the process footprints. For all expressions used to access such arrays, the statements generated by *CreateStoreAccessArray*

```
1  /* Original affine computation */
2  for( i = 0 ; i < n ; i++ )
3    for( j = 0 ; j < n ; j++ )
4      y[i] += A[i][j] * x[j];
5
6  /* Transformed parallel computation */
7  /* Update ghost read values*/
8  for( i = 0 ; i < nl ; i++ )
9    for( j = 0 ; j < n ; j++ )
10     yl[i] += Al[i][j] * xl[j];
11 /* Communicate ghost write values */
```

Listing 6. Affine conjugate gradient computation (dense matrix).

in Algorithm 1 are removed from the inspector AST. The corresponding expressions in the executor code are the affine expressions used in the original code. If loop bounds of inner loops are also affine expressions, these would be used in the executor instead of storing the loop bounds in arrays.

For all other arrays (accessed through non-affine expressions), the inspector code is used to compute a partitioning of the arrays and to create arrays that replicate the data-access patterns, as explained in Section IV. If all arrays are accessed through affine expressions, the inspector is rendered unnecessary and is discarded. Polyhedral techniques can be used to generate the executor code. Listing 6 shows the code obtained for the executor by following this approach.

VI. Evaluation

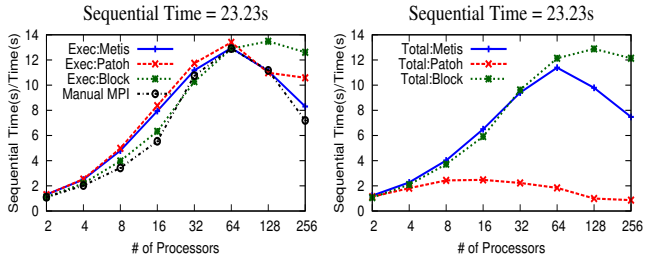
For our experimental evaluation we used a cluster with Intel Xeon E5630 processors with 4 cores per node and a clock speed of 2.67GHz, with an Infiniband interconnect. MVAPICH2-1.7 was used for MPI communications, along with Global Arrays 5.1 for the ARMCI one-sided communications. All benchmarks/applications were compiled using ICC 12.1 at -O3 optimization level.

For partitioning hypergraphs, the PaToH hypergraph partitioner [9] was used. While it supports multi-constraint hypergraph partitioning, it is sequential and requires the replication of the hypergraph on all processes. Since the generated inspector is inherently parallel, and parallel graph partitioners are available, an alternative approach was also pursued: convert the hypergraph to a graph, which can be partitioned in parallel. For this conversion, an edge was created between every pair of vertices belonging to the same net. The resulting graph was partitioned in parallel with ParMetis [40]. Multi-constraint partitioning (as discussed in Section III) was employed to achieve load balance between processes while reducing communication costs. We also evaluated a third option: block partitioning of the iterations of partitionable loops (referred to as *Block*), where the cost of graph partitioning can be completely avoided.

For all benchmarks and applications, all functions were inlined, and arrays of structures were converted to structures of arrays for use with our prototype compiler which implements the transformations described earlier. The compiler was developed within the ROSE infrastructure [36].

A. Benchmarks and Application

For evaluation purposes, we used benchmarks with data-dependent control-flow and array-access patterns. Each benchmark has a sequence of partitionable loops enclosed within an outer sequential (time or convergence) loop, with the control-flow and array-access pattern remaining the same for every iteration of that outer loop. All reported execution times are averaged over 10 runs and are normalized by the average execution time of the original sequential code.



(a) Executors, manual-MPI times (b) Inspector + Executor times

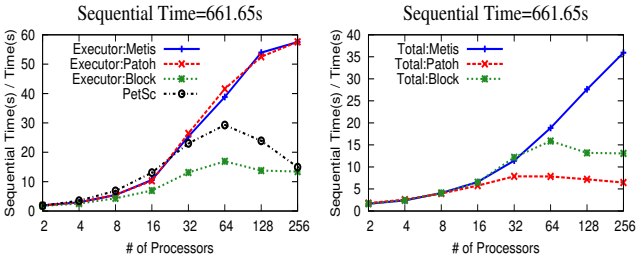
Fig. 3. 183.equake with *ref* input size

183.equake [3] is a benchmark from SPEC2000 which simulates seismic wave propagation in large basins. It consists of a sequence of partitionable loops enclosed within an outer time loop. The SPEC *ref* data size was used for the evaluation. Figure 3a compares the performance of the executor code using the three partitioning schemes with a manual MPI implementation by the authors. In all cases, the executor performance is comparable to the manual MPI implementation. After 64 processes, the performance of all executors drops off due to the overhead of communication. Figure 3b shows that the overhead of the inspector while using ParMetis or block partitioning is negligible, but with PaToH, the sequential nature of the partitioner adds considerable overhead.

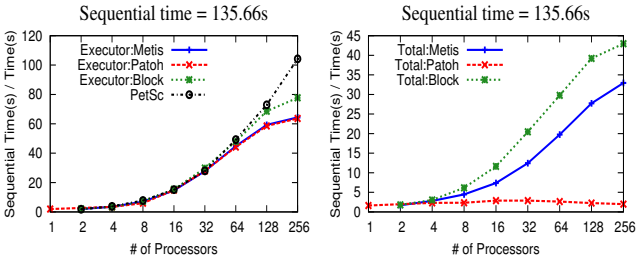
CG kernel The conjugate gradient (CG) method to solve linear system of equations consists of several partitionable loops within a convergence loop. Two sparse matrices, *hood.rb* and *tmt_sym.rb*, from the University of Florida Sparse Matrix Collections [12], stored in CSR format were used as inputs.

Figures 4a, 4c show that the executor code achieves good scaling overall with super-linear scaling between 16 and 64 processes, due to the partitions becoming small enough to fit in caches. Using block-partitioning gives good performance with *tmt_sym* but not for *hood*. For the latter, block partitioning results in higher number of ghosts cells and therefore higher communication costs, demonstrating the need for modeling the iteration-data affinity. The inspector overheads reduce the overall speed-up achieved, as shown in Figures 4b, 4d. This cost could be further amortized in cases where the linear systems represented by the matrices are solved repeatedly, say within an outer time loop, with the same non-zero structure. Such cases are common in many scientific applications.

The performance of the executors was also compared to a manual implementation using PETSc [2] which employed a block-partitioning of the rows of the matrix. For *hood*, Figure 4a shows that the performance of the generated executor code while using PaToH and ParMetis out-performs the manual PETSc implementation. The performance of the latter drops off due to the same reason the performance of the block-partitioned scheme drops off. The generated executors

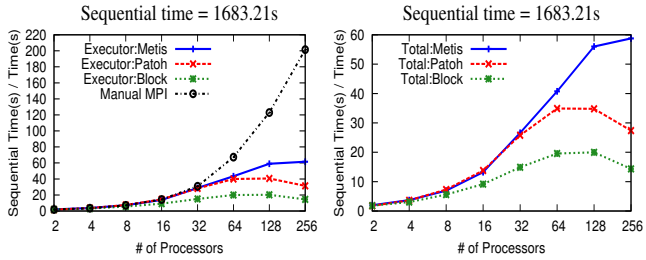


(a) Executors times - *hood.rb* (b) Total times - *hood.rb*



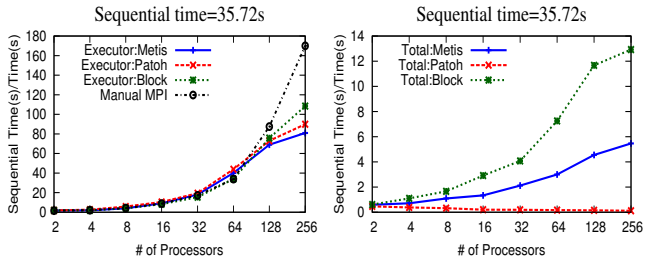
(c) Executor times - *tmt_sym.rb* (d) Total times - *tmt_sym.rb*

Fig. 4. CG Kernel with *hood.rb* and *tmt_sym.rb*



(a) Executors, manual-MPI times (b) Inspector + Executor times

Fig. 5. P3-RTE on unstructured mesh



(a) Executors, manual-MPI times (b) Inspector + Executor times

Fig. 6. miniFE-1.1 with 100x100x100 grid

perform better than the manual implementation for *hood* when using PaToH and ParMetis for partitioning. With *tmt_sym*, the generated executor performs on par with the manual implementation for all three partitioning schemes (Figure 4c).

P3-RTE [35] This benchmark solves the radiation transport equation (RTE) [27] approximated using spherical harmonics on an unstructured physical grid of 164540 triangular cells. The Finite-Volume Method is used for discretizing the RTE. Jacobi method is used for solving the system of equations at each cell center. The different partitionable loops iterate over cells, faces, nodes, and boundaries of the domain, and are enclosed within a convergence loop.

Figure 5a compares the executor times for the three schemes with a manual MPI implementation which uses domain decomposition of the underlying physical grid to partition the computation. The results once again show that a simple block-partitioning could result in poor performance. Surprisingly, ParMetis performs better than PaToH for higher number of processes. The executor code while using PaToH or ParMetis achieves performance comparable to the manual MPI implementation up to 64 processes. Past that, the manual implementation achieves super-linear scaling since it significantly reduces communication costs by replicating some computation. Using techniques like overlapping communication with computation could improve the performance of the generated executor for higher number of processes. Figure 5b shows that the inspector overhead is negligible even when using the sequential PaToH partitioner.

miniFE-1.1 [18] This is a mini-application from the Mantevo suite from Sandia National Laboratories [25]. It uses an implicit finite-element method over an unstructured 3D mesh. A problem size of 100 points along each axes was used for the evaluation. The suite also provides a manual MPI implementation of the computation. We provide a comparison of the performance of the automatically parallelized version against this manually parallelized version.

Figure 6 compares the running times of the executors (using ParMetis, PaToH, and block-partitioning) with the execution time for the manual MPI implementation. Up to 128 processes, the performance of the auto-generated executor is on par with the manual implementation. For 256 processes, the block-partitioned version performs slightly better since the manual implementation uses an approach similar to block partitioning. Figure 6b shows the speed-up achieved for the total running times. Since the actual running time of the executor is not very significant even for the large problem size, the cost of the inspector dominates the overall running time.

OLAM [45] OLAM (Ocean, Land, and Atmosphere Modeling) is an application used for climate simulations of the entire planet. It employs finite-volume methods of discretization to solve for physical quantities such as pressure, temperature, and wind velocity over an 3D unstructured grid consisting of 3D prisms covering the surface of the earth. Physical quantities are associated with centers of prisms and prism edges. The input grid contained 155520 prisms. The neighborhood information of points is stored in indirection arrays. OLAM is written in Fortran 90. Unlike the other C benchmarks, which could be fully automatically transformed by our compiler, the generation of the inspector/executor code for OLAM required some manual steps in going from the sequential application to the code generated by the compiler.

We report performance on an atmospheric model simulation consisting of 13 partitionable loops enclosed within a sequential time loop. While the outer loop typically executes hundreds of thousands of iterations, Figure 7 shows data for 30000 iterations. The time for inspection, even with a sequential hypergraph partitioner, is several orders of magnitude

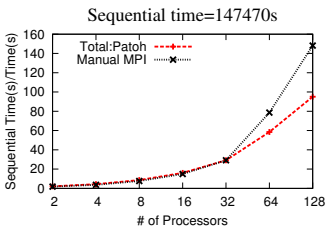


Fig. 7. OLAM atmospheric model.

lower than the executor time. Therefore, the PaToH hypergraph partitioner was used for generating the partitions.

Figure 7 shows that the performance of the code using the I/E framework (including inspector time) is on par with the manual MPI implementation. While the former scales linearly to 128 processes, the latter achieves super-linear scaling due to domain specific optimizations by experts.

VII. Related Work

Saltz and co-workers [10], [11], [38], [39] proposed the inspector-executor (I/E) approach for distributed-memory code generation for scientific applications with irregular access patterns. The PARTI/CHAOS libraries [6], [31] facilitated manual development of parallel message-passing code. Compiler support for optimizing communications within the executor was also explored [1], [44]. An approach to automatic compiler transformation for generation of I/E code via slicing analysis was developed [11], but required indirect access of all arrays in the executor code even when the original sequential code used direct access through inner loop iterators (for example, `A[j]` in Listing 1). Most of these approaches could handle only one level of indirection. The approach of Das et al. [11] could handle multiple levels of indirection, but their techniques are inapplicable to codes such as 183.equake and P3-RTE. Lain [20], [21] exploited contiguity within irregular accesses to reduce communication costs and inspector overheads. Since the layout of data was not explicitly handled to maintain contiguity, the extent to which this property could be exploited in the executor depended on the partitioning of data.

Some later approaches have proposed the use of run-time reordering transformations [13], [17], [26]. Strout et al. [41], [42] proposed a framework for code generation that combined run-time and compile-time reordering of data and computation. The recent work of LaMielle and Strout [22] proposes an extended polyhedral framework that can generate transformed code (using inspector/executor) for computations involving indirect array accesses. The class of computations addressed by that framework is more general than the partitionable loops considered here and can handle more general types of iteration/data reorderings. But the generality of the framework, without additional optimizations, can result in code that is less efficient than that generated for partitionable loops here. For example, restricting the order of execution of inner loops within partitionable loops to be the same as that of the original sequential code enables exploitation of contiguity in data access. Arbitrary iteration reordering would require use of indirect access for all expressions in the executor code. Formulation of such domain/context specific constraints within the sparse polyhedral framework so as to generate more efficient code is an interesting open question.

Basumallik and Eigenmann [5] presented techniques for translating OpenMP programs with irregular accesses into code for distributed-memory machines, by focusing on exploiting overlap of computation and communication. But the approach requires partial replication of shared data on all processes.

A large body of work has considered the problem of loop parallelization. Numerous advances in automatic parallelization and tiling of static control programs with affine array accesses have been reported [8], [14], [15], [19], [24], [32]. For loops not amenable to static analysis, speculative techniques have been used for run-time parallelization [23], [33], [37], [46]. Zhuang et al. [47] inspect run-time dependences to check if contiguous sets of loop iterations are dependent. None of those efforts address distributed memory code generation.

In contrast to prior work, we develop a framework for effective message-passing code generation and effective parallel execution of an extended class of affine computations with some forms of indirect array accesses. We are not aware of any other compiler work on inspector-executor code generation that maximize contiguity of accesses in the generated code, which is important in reducing cache misses as well as enabling SIMD optimizations in later compiler passes.

VIII. Conclusion

Irregular and sparse computations in distributed memory environments are of significant importance in scientific and engineering computing. When control-flow and array-access patterns depend on run-time data, static compiler techniques need to be combined with run-time inspection. We propose an inspector/executor parallelization approach for a class of such applications. The inspector gathers run-time information about control flow and array accesses, partitions the computation, and remaps the data and control structures. Several optimizations are used to exploit contiguity of array accesses. Experimental results show that the generated code comes close to the performance achieved by manual parallelization by domain experts. Future work would focus on optimizing the communication layer to achieve better scalability of the generated code.

IX. Acknowledgments

We thank Umit Catalyurek for making available the PaToH software and for his help with formulating the partitioning constraints. We thank Gagan Agrawal and the SC reviewers for their comments and feedback with respect to existing inspector/executor techniques. We also thank Robert L. Walko, the author of OLAM, for his guidance with the software. This material is based upon work supported by the U.S. National Science Foundation under grants 0811457, 0811781, 0926687, 0926688 and 0904549, by the U.S. Department of Energy under grant DE-FC02-06ER25755, and by the U.S. Army through contract W911NF-10-1-0004.

REFERENCES

[1] G. Agrawal, J. Saltz, and R. Das, "Interprocedural partial redundancy elimination and its application to distributed memory compilation," in *PLDI*, 1995.

[2] S. Balay, J. Brown, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, "PETSc Web page," 2012, http://www.mcs.anl.gov/petsc.

[3] H. Bao, J. Bielak, O. Ghattas, L. F. Kallivokas, D. R. O'Hallaron, J. R. Shewchuk, and J. Xu, "Large-scale simulation of elastic wave propagation in heterogeneous media on parallel computers," *Computer Methods in Applied Mechanics and Engineering*, vol. 152, pp. 85–102, 1998.

[4] M. M. Baskaran, J. Ramanujam, and P. Sadayappan, "Automatic C-to-CUDA code generation for affine programs," in *CC*, 2010.

[5] A. Basumallik and R. Eigenmann, "Optimizing irregular shared-memory applications for distributed-memory systems," in *PPoPP*, 2006.

[6] H. Berryman, J. Saltz, and J. Scroggs, "Execution time support for adaptive scientific algorithms on distributed memory machines," *Concurrency: Practice and Experience*, vol. 3, pp. 159–178, 1991.

[7] U. Bondhugula, O. Gunluk, S. Dash, and L. Renganarayanan, "A model for fusion and code motion in an automatic parallelizing compiler," in *PACT*, 2010.

[8] U. Bondhugula, A. Hartono, J. Ramanujan, and P. Sadayappan, "A practical automatic polyhedral program optimization system," in *PLDI*, 2008.

[9] U. V. Catalyurek and C. Aykanat, *PaToH: Partitioning Tool for Hypergraphs*, 2009.

[10] R. Das, P. Havlak, J. Saltz, and K. Kennedy, "Index array flattening through program transformation," in *SC*, 1995.

[11] R. Das, J. Saltz, and R. von Hanxleden, "Slicing analysis and indirect access to distributed arrays," Rice University, Tech. Rep. CRPC-TR93319-S, 1993.

[12] T. A. Davis, "University of Florida sparse matrix collection," *NA Digest*, 1994.

[13] C. Ding and K. Kennedy, "Improving cache performance in dynamic applications through data and computation reorganization at run time," in *PLDI*, 1999.

[14] P. Feautrier, "Some efficient solutions to the affine scheduling problem - Part I: One-dimensional time," *IJPP*, vol. 21, pp. 313–347, 1992.

[15] M. Griebl, *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. FMI, University of Passau, 2004.

[16] M. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, and M. M. Khan, "Loop transformation recipes for code generation and auto-tuning," in *LCPC*, 2010.

[17] H. Han and C.-W. Tseng, "Exploiting locality for irregular scientific codes," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, pp. 606–618, 2006.

[18] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, 2009.

[19] F. Irigoin and R. Triolet, "Supernode partitioning," in *POPL*, 1988.

[20] A. Lain, "Compiler and run-time support for irregular computations," Ph.D. dissertation, University of Illinois at Urbana-Champagne, 1996.

[21] A. Lain and P. Banerjee, "Exploiting spatial regularity in irregular iterative applications," in *IPPS*, 1995.

[22] A. LaMielle and M. Strout, "Enabling code gen. with sparse polyhedral framework," Colorado State University, Tech. Rep. CS-10-102, 2010.

[23] S.-T. Leung and J. Zahorjan, "Improving the performance of runtime parallelization," in *PPoPP*, 1993.

[24] A. W. Lim and M. S. Lam, "Maximizing parallelism and minimizing synchronization with affine transforms," in *POPL*, 1997.

[25] "Mantevo project," `https://software.sandia.gov/mantevo`.

[26] N. Mitchell, L. Carter, and J. Ferrante, "Localizing non-affine array references," in *PACT*, 1999.

[27] M. F. Modest, *Radiative Heat Transfer*. Academic Press, 2003.

[28] J. Neiplocha, V. Tipparaju, M. Krishnan, and D. K. Panda, "High performance remote memory access communication: The ARMCI approach," *Int. J. High Performance Computing Applications*, vol. 20, pp. 233–253, 2006.

[29] "Par4all," `www.par4all.org`.

[30] "Pluto - an automatic parallelizer and locality optimizer for multicores," `pluto-compiler.sourceforge.net`.

[31] R. Ponnusamy, J. H. Saltz, and A. N. Choudhary, "Runtime compilation techniques for data partitioning and communication schedule reuse," in *SC*, 1993.

[32] J. Ramanujam and P. Sadayappan, "Tiling multidimensional iteration spaces for multicomputers," *JPDC*, vol. 16, no. 2, pp. 108–230, 1992.

[33] L. Rauchwerger and D. Padua, "The LRPD test: Speculative runtime parallelization of loops with privatization and reduction parallelization," in *PLDI*, 1995.

[34] M. Ravishankar, J. Eisenlohr, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan, "Code generation for parallel execution of a class of irregular loops on distributed memory systems," The Ohio State University, Tech. Rep. OSU-CISRC-5/12-TR10, 2012.

[35] M. Ravishankar, S. Mazumder, and A. Kumar, "Finite-volume formulation and solution of the p3 equations of radiative transfer on unstructured meshes," *Journal of Heat Transfer*, vol. 132, p. 0123402, 2010.

[36] "Rose compiler infrastructure," `www.rosecompiler.org`.

[37] S. Rus, M. Pennings, and L. Rauchwerger, "Sensitivity analysis for automatic parallelization on multi-cores," in *ICS*, 2002.

[38] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman, "Run-time scheduling and execution of loops on message passing machines," *J. Parallel Distrib. Comput.*, vol. 8, pp. 303–312, 1990.

[39] J. H. Saltz, H. Berryman, and J. Wu, "Multiprocessors and run-time compilation," *Concurrency: Practice and Experience*, 1991.

[40] K. Schloegel, G. Karypis, and V. Kumar, "Parallel static and dynamic multi-constraint graph partitioning," *Concurrency and Computation: Practice and Experience*, vol. 14, pp. 219–240, 2002.

[41] M. M. Strout, L. Carter, and J. Ferrante, "Compile-time composition of run-time data and iteration reodrderings," in *PLDI*, 2003.

[42] M. M. Strout, L. Carter, J. Ferrante, J. Freeman, and B. Kreaseck, "Combining performance aspects of irregular Gauss-Seidel via sparse tiling," in *LCPC*, 2002.

[43] M. M. Strout and P. D. Hovland, "Metrics and models for reordering transformations," in *MSP*, 2004.

[44] R. von Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz, "Compiler analysis for irregular problems in Fortran D," *LCPC*, 1993.

[45] R. L. Walko and R. Avissar, "The Ocean-Land-Atmosphere Model (OLAM). Part I: Shallow-Water Tests," *Monthly Weather Review*, vol. 136, pp. 4033–4044, 2008.

[46] H. Yu and L. Rauchwerger, "Techniques for reducing the overhead of run-time parallelization," in *CC*, 2000.

[47] X. Zhuang, A. E. Eichenberger, Y. Luo, K. O'Brien, and K. O'Brien, "Exploiting parallelism with dependence-aware scheduling," in *PACT*, 2009.